ETSI TS 103 744 V1.2.2 (2026-02)

TECHNICAL SPECIFICATION

CYBER;
Quantum-Safe Cryptography (QSC);
Quantum-safe Hybrid Key Establishment

Reference
RTS/CYBER-QSC-0031

Keywords
key exchange, quantum safe cryptography

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI IPR online database.

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™**, **LTE™** and **5G™** logo are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

Foreword

This Technical Specification (TS) has been produced by ETSI Technical Committee Cyber Security (CYBER).

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

Introduction

Hybrid Key Establishments are constructions that combine a traditional key establishment method, such as elliptic curve Diffie Hellman [1], with a quantum-safe key encapsulation mechanism, such as Module-Lattice-based Key Encapsulation Mechanism (ML-KEM) [11], into a single key establishment method. Hybrid key establishments are a migration technique to move to quantum-safe technology in advance of establishing full security assurance in the underlying post-quantum cryptographic scheme.

1 Scope

The present document specifies several methods for deriving cryptographic keys from multiple shared secrets. The shared secrets are established using existing traditional key establishment schemes, like Elliptic Curve Diffie-Hellman (ECDH) in NIST SP800-56Ar3 [1], and new quantum-safe Key Encapsulation Mechanisms (KEMs).

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found in the ETSI docbox.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are necessary for the application of the present document.

[1] NIST SP800-56Ar3: "Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography".

[2] IETF RFC 2104: "HMAC: Keyed-Hashing for Message Authentication".

[3] IETF RFC 5869: "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)".

[4] FIPS PUB 180-4: "Secure Hash Standard (SHS)".

[5] Void.

[6] NIST SP800-186: "Recommendations for Discrete Logarithm-Based Cryptography: Elliptic Curve Domain Parameters".

[7] IETF RFC 5639: "Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation".

[8] IETF RFC 7748: "Elliptic Curves for Security".

[9] NIST SP800-185: "SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash".

[10] NIST SP800-56Cr2: "Recommendation for Key-Derivation Methods in Key-Establishment Schemes".

[11] FIPS 203: "Module-Lattice-Based Key-Encapsulation Mechanism Standard".

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] Y. Dodis, R. Gennaro, J. Håstad, H. Krawczyk, and T. Rabin: "Randomness Extraction and Key derivation Using the CBC, Cascade, and HMAC Modes", Crypto 04, LNCS 3152, pp. 494-510. Springer Verlag, 2004.

[i.2] Void.

[i.3] N. Bindel, J. Brendel, M. Fischlin, B. Goncalves, D. Stebila: "Hybrid Key Encapsulation Mechanisms and Authenticated Key Exchange", IACR eprint 2018-903.

[i.4] Void.

[i.5] Simon D. R.: "On the power of quantum computation", SFCS 94 Proceedings of the 35th Annual Symposium on Foundations of Computer Science, November 1994, Pages 116-123.

[i.6] Shor P.W.: "Algorithms for quantum computation: discrete logarithms and factoring", SFCS 94: Proceedings of the 35th Annual Symposium on Foundations of Computer Science, November 1994, Pages 124-134.

[i.7] Void.

[i.8] Void.

[i.9] Void.

[i.10] Void.

[i.11] Campagna M., Petcher A.: "Security of Hybrid Key Encapsulation", IACR eprint 2020-1364, November 2020.

[i.12] Campagna M., Petcher A.: "Security of Hybrid Key Establishment using Concatenation", IACR eprint 2023-972, June 2023.

[i.13] Void.

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the following terms apply:

asymmetric cryptography: cryptographic system that utilizes a pair of keys, a private key known only to one entity, and a public key that can be openly distributed without loss of security

big-endian: octet ordering that signifies "big-end", or most significant octet value is stored to the left, or at the lowest storage location

EXAMPLE: The decimal value 108591, which is 0x0001A82F as a hex encoded 32-bit integer, is encoded as a length 4 octet string as 0001A82F.

cryptographic hash function: function that maps a bit string of arbitrary length to a fixed length bit string (*message digest* or *digest* for short)

NOTE: Hash functions are designed to satisfy the following properties:

1) (One-way) It is computationally infeasible to find any input that maps to any pre-specified output.

2) (Collision resistant) It is computationally infeasible to find any two distinct inputs that map to the same output.

cryptographic key: binary string used as a secret by a cryptographic algorithm

EXAMPLE: AES-256 requires a random 256-bit string as a secret key.

entity: person, device, or system that is executing the steps of a process

NOTE: Steps of one of the processes defined or referenced in the present document.

info: octet string set by the application as additional information

EXAMPLE: An application specific value like an ASCII encoded string,
e.g. info = "ETSI_QSHKE_TEST_VECTORS_V_1_2".

key agreement scheme: key establishment procedure in which the resultant secret keying material is a function of contributions of the entities participating, such that no entity can predetermine the value of the secret keying material independently of the other entities' contributions

key derivation: process to derive key material from one or more shared secrets

key encapsulation mechanism: set of methods to establish a shared secret key between two parties

key establishment/exchange method: cryptographic procedure by which cryptographic keys are established between two parties

label: octet string that specifies a separation of use for the instance of the key derivation or exchange, such as a random nonce

message digest/digest: fixed-length output of a cryptographic hash function over a variable length input

octet string: ordered sequence of octets/bytes consisting of 8-bits each

private key: key in an asymmetric cryptographic scheme that is kept secret

public key: key in an asymmetric cryptographic scheme that can be made public without loss of security

public key cryptography: See asymmetric cryptography.

random oracle: theoretical black box that responds to every unique query with a uniformly random selection from the set of possible responses, with repeated queries receiving the same response

security level: value n for which the best-known attack against breaking the security properties of a cryptographic algorithm requires 2^n operations.

NOTE: Sometimes also referred to as *bit-strength*.

shared secret: secret value that has been computed using a key-establishment scheme

3.2 Symbols

For the purposes of the present document, the following symbols apply:

A || B	The concatenation of binary strings A followed by B
$\varnothing$	A zero-length octet string
$[x]_n$	An integer value x expressed as an n-bit integer
$\lceil q \rceil$	The least integer value x greater than or equal to q
len(A)	The number of octets in an octet string *A*
hash( )	A cryptographic hash function
digest_len	The length in octets of a hash function's digest
block_len	The block length in octets of a hash function's block size
C	A ciphertext value created by a KEM
d	A private key for elliptic curve cryptography
k	A cryptographic secret or key
P/R	A public key for an asymmetric cryptographic scheme
psk	A pre-shared key
Q	A public key for elliptic curve cryptography

sk A private key for an asymmetric cryptographic scheme

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

AES	Advanced Encryption Standard
BP-P	BrainPool elliptic curve over finite Prime field
CAVP	Cryptographic Algorithm Validation Program
CDH	Cofactor Diffie-Hellman
CID	Ciphersuite IDentifier
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDHE	Elliptic Curve Diffie-Hellman Ephemeral
HKDF	HMAC-based Key Derivation Function
HMAC	Hash-based Message Authentication Code
IND-CCA	INDistinguishability under Chosen-Ciphertext Attacks
IND-CPA	INDistinguishability under Chosen-Plaintext Attacks
KDF	Key Derivation Function
KEM	Key Encapsulation Mechanism
KMAC	Keccak Message Authentication Code
LNCS	Lecture Notes in Computer Science
MA	Message from entity A
MB	Message from entity B
ML-KEM	Module-Lattice-Based Key Encapsulation Mechanism
NIST	National Institute of Standards and Technology
OW-CCA	One Way Chosen Ciphertext Attack
OW-CPA	One-Way Chosen-Plaintext Attack
PRF	PseudoRandom Function
QKD	Quantum Key Distribution
RSA	Rivest, Shamir and Adelman
SP	Special Publication
SSH	Secure Shell
TLS	Transport Layer Security

4 Purpose of quantum-safe hybrid key establishment

4.1 Status of quantum-safe key encapsulation mechanisms

NIST has initiated a process of analysing and standardizing one or more new quantum-safe key encapsulation mechanisms suitable to replace traditional key establishment schemes. At the time of the present document, there is one FIPS approved standard, FIPS 203 [11].

The present document addresses the following cases:

1) One or more key exchange method establishes a shared secret from which randomness extraction is necessary.

2) One or more key exchange method incorporates a hash-based key derivation function prior to use within the hybrid method defined in the present document.

Quantum-safe hybrid key establishment specified in the present document ensures that the derived key is at least as secure as the maximum security of the key establishment schemes. The resulting hybrid scheme will remain secure if one of the key establishment schemes remains secure.

Quantum Key Distribution (QKD) provides an alternative method of establishing a shared secret between two entities using quantum mechanics. The scope of the present document is limited to elliptic curve Diffie-Hellman and quantum-safe key encapsulation mechanisms.

5 Architecture for quantum-safe hybrid key establishment

5.1 Functional entities

There are two entities defined for quantum-safe hybrid key establishment, an Initiator *A* that initiates a key establishment scheme, and a Responder *B* who responds to the request. The entities communicate over a network medium.

EXAMPLE: Examples of such mediums are: ethernet, wireless and cellular networks.

Figure 1: Communicating entities *A* and *B*

5.2 Information relationships (reference points)

The network media over which the Initiator and Responder communicate will have a packet formatting scheme that allows the encoding and transmission of octet (byte) strings. The Initiator and Responder will exchange messages, where each message is an octet string that can span multiple packets. *MA* denotes a message from *A* to *B*, and *MB* denotes a message sent from *B* to *A*.

A may initiate a hybrid key establishment by the transmission of a message to *B*. *B* responds to this message. The exchange between the entities can consist of a single message or multiple rounds of messages.

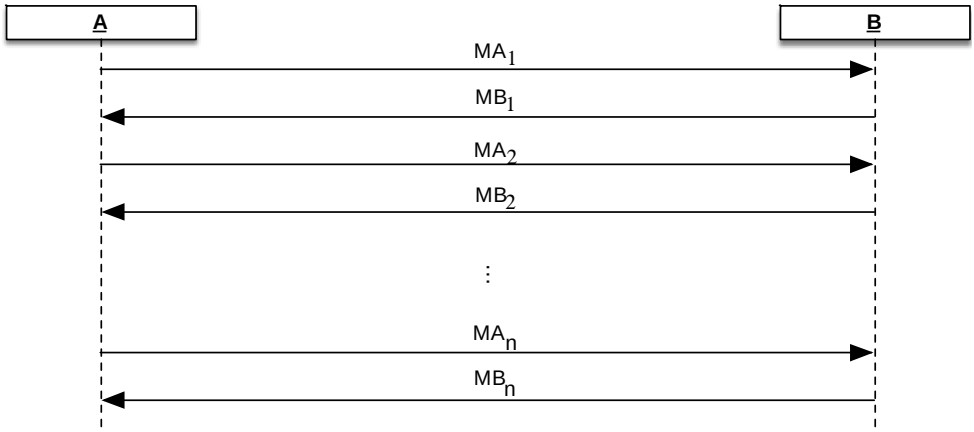

Figure 2: Messages exchanged between entities *A* and *B*

The transcript of the key establishment is the list of all messages exchanged between *A* and *B*, in the sequence order they were sent:

$$transcript = (MA_1, MB_1, MA_2, MB_2, \ldots MA_n, MB_n)$$

In other embodiments, *B* may be in possession of authentic public keys belonging to *A*. The exchange of messages may consist solely of messages from *B* to *A*.

6 Introductory information

6.1 Introduction

Quantum-safe hybrid key establishment combines a traditional key establishment scheme, like ECDH and a quantum-safe Key Encapsulation Mechanism (KEM). Hybrid key establishment schemes specified in the present document use two or more shared secrets to derive cryptographic key material using a key derivation function. The key derivation functions for hybrid key establishment specified in the present document provide both the key expansion property and random extraction as per Crypto 04, LNCS 3152 [i.1].

6.2 Notation

6.2.1 Radix

The prefix "0x" indicates hexadecimal numbers.

6.2.2 Conventions

The assignment operator "=", as used in several programming languages:

<variable> = <expression>

means that <*variable*> assumes the value that <*expression*> had before the assignment took place. For instance:

$$x = x + y + 3$$

means:

(new value of x) becomes (old value of x) + (old value of y) + 3.

6.2.3 Bit/Byte ordering

All data variables are represented with the most significant bit (or byte) on the left-hand side and the least significant bit (or byte) on the right-hand side. Where a variable is broken down into a number of sub-strings, the left most (most significant) sub-string is numbered 0, the next most significant is numbered, 1 and so on, through to the least significant.

EXAMPLE: An n-bit MESSAGE is subdivided into 64-bit substrings M_0, M_1, …, M_i so if the message is:

0x0123456789ABCDEFFEDCBA98765432 1086545381AB594FC28786404C50A37…

then:

M_0 = 0x0123456789ABCDEF
M_1 = 0xFEDCBA9876543210
M_2 = 0x86545381AB594FC2
M_3 = 0x8786404C50A37…

6.2.4 Integer encoding

Integers are represented in the bit/byte ordering defined in clause 6.2.3. The most significant bit (or byte) on the left-hand side and the least significant bit (or byte) on the right-hand side.

EXAMPLE: A 32-bit integer of the value I = 37 is encoded as:

I = 0x00000025

NOTE: This is big-endian or network byte ordering.

7 Cryptographic primitives

7.1 Hash functions (hash)

A hash function maps an arbitrary length bit string *(input)* to a fixed length (*digest_len*) octet string output *(digest)*:

$$digest = hash(input)$$

Approved hash functions for the purpose of the present document shall be limited to those in the following list:

- SHA-256, SHA-384 as defined in FIPS PUB 180-4 [4].

7.2 Context formatting function (*f*)

7.2.1 Context formatting function (*f*) description

The context formatting functions used in the present document take a list of inputs and return an octet string. A generic calling interface to the context function *f* used in the present document is defined in the present clause:

$$context = f(val1, val2,\ldots)$$

where the parameters and output shall be defined as follows:

Input:

$val_1, val_2, \ldots, val_n$ - an ordered sequence of octet strings each of length less than 2^{32} octets, where $n > 0$.

Output:

context - an octet string representing the context value.

7.2.2 Concatenate-based context formatting function

The present clause defines a concatenate-based context formatting function. The concatenate-based context formatting function takes an ordered sequence of octet strings and converts them into a length-delimited single octet string. The concatenate-based context formatting function *f* has the following calling interface:

$$context = cb\_f(val_1, val_2, \ldots)$$

where the parameters, procedure and output shall be as follows:

Input:

$val_1, val_2, \ldots, val_n$ - an ordered sequence of octet strings each of length less than 2^{32} octets, where $n > 0$.

Process:

1) *Set context = ∅.*
2) For $i = 1, \ldots, n$.
   a) Set $len_i = len(val_i)$, returns the length of val_i in octets.
      i) If $len_i > 2^{32} - 1$, return error.
      ii) $L_i = [len_i]_{32}$ - a 32-bit integer value expressed as 4 octets.
   b) Set $context = context \,||\, L_i \,||\, val_i$.
3) Return *context*.

Output:

context - an octet string representing the context value or an error.

The concatenate-based formatting function is designed to be used in conjunction with the KMAC variety of KDFs. HKDF implementations do not perform efficiently when instantiated with a concatenate-based context formatting function. The *context* value is passed as the 'info' input to the HKDF function. See OpenSSL's documentation on length limits for *context* - https://www.openssl.org/docs/man1.1.1/man3/EVP_PKEY_CTX_add1_hkdf_info.html. IETF RFC 5869 [3] does not limit the length of the 'info' input value; however, for efficiency reasons, it is recommended to keep the length relatively short since it is processed through a hash function in every execution of HKDF expand step.

7.2.3 Concatenate-and-hash-based context formatting function

The present clause defines a concatenate-and-hash context formatting function. The concatenate-and-hash context formatting function takes an ordered sequence of octet strings and converts them into a length delimited octet string. The concatenate-and-hash-based context formatting function *f* has the following calling interface:

$$context = cahb\_f(val_1, val_2, \ldots)$$

where the prerequisite, parameters, procedure and output shall be as follows:

Prerequisite:

hash - an approved hash function as per clause 7.1 that produces a *digest_len*-length string of octets (*digest_len* in {32, 48}).

Input:

$val_1, val_2, \ldots, val_n$ - an ordered sequence of octet strings each of length less than 2^{32} octets, where $n > 0$.

Process:

1) Set $buffer = \varnothing$.
2) For $i = 1, \ldots, n$.
   a) Set $len_i = len(val_i)$, returns the length of val_i in octets.
      i) If $len_i > 2^{32} - 1$, return error.
      ii) $L_i = [len_i]_{32}$ - a 32-bit integer value expressed as 4 octets.
   b) Set buffer = buffer || L_i || val_i.
3) Set $context = hash(buffer)$.
4) Return context.

Output:

context - a *digest_len* octet string representing the context value or an error.

NOTE: In practice, the concatenate-and-hash based context format function may save memory allocation, and make use of the *context* value more efficiently if there are cases where it is used iteratively.

7.3 PseudoRandom Function (PRF)

7.3.1 PRF description

A PseudoRandom Function generates output from a random (or secret) seed such that the output is computationally indistinguishable from truly random output. A generic calling interface to the PRF used in the present document is defined in the present clause:

$$output = PRF(secret, val_1, val_2, \ldots)$$

where the parameters and output shall be defined as follows:

Input:

secret - an octet string that constitutes the input for the pseudorandom function.

$val_1, val_2, \ldots, val_n$ - an ordered sequence of octet strings each of length less than 2^{32} octets, where $n > 0$.

Output:

output - the pseudorandom output, length dependent on the primitive used in the PRF.

Approved PRF functions for the purpose of the present document shall be limited to HMAC as defined in IETF RFC 2104 [2] with a SHA2 hash function from clause 7.1, or KMAC128 or KMAC256 as defined in NIST SP800-185 [9].

7.3.2 PRF to HMAC mapping

This clause specifies a mapping from the PRF definition to HMAC with a specified hash function.

HMAC shall be as defined in IETF RFC 2104 [2] with an approved SHA2 hash function from clause 7.1 and has a calling interface:

$$output = HMAC(secret, data)$$

where the prerequisite, parameters, and output shall be defined as follows:

Prerequisite:

cahb_f - a context formatting function as defined in clause 7.2.3.

Input:

secret - an octet string that constitutes the input for the pseudorandom function.

data - an octet string that is included in the HMAC.

Output:

output - pseudorandom output of a fixed length (of the underlying hash function).

The mapping shall be defined as follows:

$$output = PRF\ (secret, val_1, val_2, \ldots) = HMAC(secret, cahb\_f(val_1, val_2, \ldots)).$$

7.3.3 PRF to KMAC mapping

This clause specifies a mapping from the PRF definition to KMAC function, KMAC128 or KMAC256. KMAC shall be as defined in NIST SP800-185 [9] and has a calling interface:

$$output = KMAC(K, X, L, S)$$

where the prerequisite, parameters, and output shall be defined as follows:

Prerequisite:

cb_f - a context formatting function as defined in clause 7.2.2.

Input:

K - a key bit string of any length, at least security strength in length.

X - an input bit string of any length.

L - is an integer representation of the output length in bits.

S - is an optional customization bit string of any length.

Output:

output - pseudorandom output of bit length *L*.

The mapping shall be defined as follows:

$$output = PRF (secret, val_1, val_2, \ldots) = KMAC (secret, cb\_f(val_1, val_2, \ldots), kmac\_outlen, \varnothing),$$

where *kmac_outlen* is set to 32 or 48 octets for KMAC128 and KMAC256, respectively, corresponding to the minimum security level of the underlying key establishment schemes, and ∅ represents an empty octet string.

If a *secret* value is not specified or is the empty string, then a *secret* value of a zero string of 164 octets shall be used for KMAC128, and a *secret* value of a zero string of 132 octets shall be used for KMAC256. This shall only happen in the case when a *psk* is not used in the cascade combiner, see clause 8.3.3.

7.4 Key Derivation Functions (KDFs)

7.4.1 KDF description

The key derivation functions used in the present document are derived from PRFs. A generic calling interface to the KDFs used in the present document is defined in the present clause:

key_material = KDF(secret, label, context, length)

where the parameters and output shall be defined as follows:

Input:

secret - an octet string that constitutes the input for the key derivation function.

label - an optional octet string that specifies a separation of use for the instance of the key derivation or exchange, such as a random nonce.

context - an octet string that constitutes an input for the key derivation function. It may include identity information specific to the entities deriving keys or exchanged messages.

length - the length in octets of the derived keying material.

Output:

key_material - the derived keying material of length *length*.

Approved KDF functions for the purpose of the present document shall be limited to HKDF as defined in IETF RFC 5869 [3] with an approved SHA2 hash function from clause 7.1, or KMAC128 or KMAC256 as defined in NIST SP800-185 [9].

NOTE: The KDFs selected for the present document are based on meeting the random oracle property used in the security proof [i.12], and their availability in existing cryptographic modules.

7.4.2 KDF to HKDF mapping

This clause specifies a mapping from the KDF definition to HKDF with a SHA2 hash function from clause 7.1.

HKDF shall be as defined in IETF RFC 5869 [3] and has a calling interface:

key_material = HKDF(secret, salt, info, length)

where the parameters and output shall be defined as follows:

Input:

secret - an octet string that constitutes the input for the key derivation function. It shall be present.

salt - optional salt value (a non-secret random value); it may be present. If it is not present it shall be set to a *digest_len* octet string of zero values. The *digest_len* is defined by the underlying hash function.

info - an octet string that contains application specific information. It may be a zero-length string.

length - the length in octets of the derived keying material. It shall be present.

Output:

key_material - the derived keying material of length *length*.

The mapping shall set *salt = label*, and info = context, as follows:

key_material = KDF(secret, label, context, length) = HKDF(secret, label, context, length).

7.4.3 KDF to HMAC mapping

This clause specifies a mapping from the KDF definition to an HMAC implementation of the NIST SP800-56Cr2 [10] One step Key Derivation Function using HMAC. HMAC shall be as defined in IETF RFC 2104 [2] with an approved SHA2 hash function from clause 7.1 and has a calling interface:

output = HMAC(secret, data)

where the prerequisite, parameters, and output shall be defined as follows:

Input:

secret - an octet string that constitutes the input for the pseudorandom function.

data - an octet string that is included in the HMAC.

Output:

output - pseudorandom output of a fixed length (of the underlying hash function).

The mapping shall set the *secret = label* and *data = counter* || *secret* || *context*, where *HMAC(label, counter* || *secret* || *context)* is iteratively executed to generate at least *length* output octets, with a 32-bit *counter* value. The streamed output is truncated (if needed) to output *key_material* of exactly *length*-octets. The mapping adheres to the following process.

Process:

1) Let $n = \lceil length/digest\_len \rceil$.
2) If $n > 2^{32} - 1$ return error.
3) If len(*secret* || *context*) > (*block_len* - 4*)* return error.
4) Set *buffer* = ∅.
5) For *counter = 1 to n*.
   a) Set *buffer = buffer* || *HMAC(label, [counter]*$_{32}$ || *secret* || *context), [counter]*$_{32}$ a 32-bit integer value expressed as 4 octets.

6) Set *key_material* = left most *length*-octets of *buffer*.

7) Return *key_material*.

If a *label* value is not specified, then a *label* value of a zero string of *block_len* octets shall be used.

7.4.4 KDF to KMAC mapping

This clause specifies a mapping from the KDF definition to KMAC implementation of the NIST SP800-56Cr2 [10] One step Key Derivation Function using KMAC128 or KMAC256. KMAC shall be as defined in NIST SP800-185 [9] and has a calling interface:

$$key\_material = KMAC(K, X, L, S)$$

where the parameters and output shall be defined as follows:

Input:

K - a key bit string of any length, at least security strength in length.

X - an input bit string of any length.

L - is an integer representation of the output length in bits.

S - is an optional customization bit string of any length.

Output:

key_material - the derived keying material of bit length *L*.

The mapping shall set $K = label$, $X = counter \,||\, secret \,||\, context$ and $L = length \times 8$, as follows:

$key\_material = KDF(secret, label, context, length) = KMAC(label, counter \,||\, secret \,||context, length \times 8, "KDF")$,

with big-endian 4-byte unsigned integer counter = 0x00000001.

If a *label* value is not specified, then a *label* value of a zero string of 164 octets shall be used for KMAC128, and a *label* value of a zero string of 132 octets shall be used for KMAC256, as specified in NIST SP800-56Cr2 [10].

NOTE 1: The present document requires the lengths of the bit strings *K*, *X* and *S* to be multiples of 8.

NOTE 2: The requirement that S = "KDF" is part of the NIST SP800-56Cr2 [10] requirement on the use of KMAC as a key derivation function.

7.5 Elliptic Curve Diffie-Hellman (ECDH)

7.5.1 ECDH description

One of the key-establishment schemes shall be Elliptic Curve Diffie-Hellman defined in clause 5.7.1.2 of NIST SP800-56Ar3 [1]. A shared secret *k* is computed between two entities over the same elliptic curve domain parameters. The key-establishment scheme, for a given set of elliptic curve domain parameters, consists of a pair of algorithms:

- *ECKeyGen( )*, a probabilistic algorithm that returns a private and public key pair *(d, Q)*.
- $ECDH(d_A, Q_B)$, a deterministic algorithm that takes d_A, entity A's private key, and Q_B, entity B's public key, as input and computes a shared secret *k*, or ⊥ an error indicator.

Similarly, party B computes the same shared secret by computing $ECDH(d_B, Q_A)$.

7.5.2 Elliptic curve domain parameters

The elliptic curve parameters used shall be one of the following:

- NIST P-384/secp384r1 defined in clause G.1.3 of NIST SP 800-186 [6].
- NIST P-256/secp256r1 defined in clause G.1.2 of NIST SP 800-186 [6].
- brainpoolP384r1 defined in clause 3.6 of IETF RFC 5639 [7].
- brainpoolP256r1 defined in clause 3.4 of IETF RFC 5639 [7].
- Curve448 defined in clause 4.2 of IETF RFC 7748 [8].
- Curve25519 defined in clause 4.1 of IETF RFC 7748 [8].

7.6 Key Encapsulation Mechanisms (KEMs)

7.6.1 KEM description

The present clause specifies the basic properties of a Key Encapsulation Mechanism (KEM) as a tuple of algorithms (*KEMKeyGen*, *Encaps*, *Decaps*) associated with a key space *K*:

- *KEMKeyGen( )*, a probabilistic algorithm that returns a private and public key pair *(sk, P)*, or ⊥ an error indicator.
- *Encaps(P)*, a probabilistic algorithm that takes a public key *P* as input and outputs a cryptographic key from the key space *K* and an associated ciphertext encapsulation using the public key *P*, denoted by *(k, C)*, or ⊥ an error indicator.
- *Decaps(sk, C)*, a deterministic algorithm that takes a private key *sk* and ciphertext *C* as input and outputs a cryptographic key *k* from the key space *K*, or a rejection key *k'*.

All KEMs shall provide OW-CPA security (see annex B).

7.6.2 Post-quantum KEMs

The present clause specifies the basic properties of post-quantum KEMs as a tuple of algorithms. All post-quantum KEMs shall expose a tuple of algorithms matching the KEM properties defined in clause 7.6.1. The current list of approved post-quantum KEMs consists of the FIPS-approved ML-KEM [11].

7.7 Primitive parameter sets

7.7.1 Parameter set description

The present clause specifies a parameter set. Parameter sets consist of tuples of specific choices for a key derivation function, elliptic curve Diffie-Hellman and quantum-safe key encapsulation mechanism instances. Parameters sets of are denoted KDF_CURVE_KEM.

Clause 7.7.2 restricts the parameter sets to enable greater interoperability.

The PRF and context formatting function are defined from the KDF function. When the KDF is instantiated as HKDFwHash or HMACwHash, the PRF shall be HMAC and the context formatting function will be *cahb_f* with the same hash function. When the KDF is KMAC#, the PRF shall be KMAC# and the context formatting function will be *cb_f*.

7.7.2 Parameter sets

The present clause specifies sets of primitives. The parameter sets are defined by a triple, a key derivation mechanism, a first key establishment scheme, and a second key establishment scheme. An implementation shall use one of the following parameter sets.

HKDFwSHA256_P256_ML-KEM-512,

HKDFwSHA256_ X25519_ML-KEM-512,

HKDFwSHA256_BP-P256_ML-KEM-512,

HMACwSHA256_P256_ML-KEM-512,

HMACwSHA256_ X25519_ML-KEM-512,

HMACwSHA256_BP-P256_ML-KEM-512,

KMAC128_P256_ML-KEM-512,

KMAC128_ X25519_ML-KEM-512,

KMAC128_ BP-P256_ML-KEM-512,

HKDFwSHA256_P256_ML-KEM-768,

HKDFwSHA256_ X25519_ML-KEM-768,

HKDFwSHA256_ BP-P256_ML-KEM-768,

HMACwSHA256_P256_ML-KEM-768,

HMACwSHA256_ X25519_ML-KEM-768,

HMACwSHA256_ BP-P256_ML-KEM-768,

KMAC128_P256_ML-KEM-768,

KMAC128_ X25519_ML-KEM-768,

KMAC128_ BP-P256_ML-KEM-768,

HKDFwSHA384_P384_ML-KEM-768,

HKDFwSHA384_ X448_ML-KEM-768,

HKDFwSHA384_ BP-P384_ML-KEM-768,

HMACwSHA384_P384_ML-KEM-768,

HMACwSHA384_ X448_ML-KEM-768,

HMACwSHA384_ BP-P384_ML-KEM-768,

KMAC256_P384_ML-KEM-768,

KMAC256_ X448_ML-KEM-768,

KMAC256_ BP-P384_ML-KEM-768,

HKDFwSHA384_P384_ML-KEM-1024,

HKDFwSHA384_ X448_ML-KEM-1024,

HKDFwSHA384_ BP-P384_ML-KEM-1024,

HMACwSHA384_P384_ML-KEM-1024,

HMACwSHA384_ X448_ML-KEM-1024,

HMACwSHA384_ BP-P384_ML-KEM-1024,

KMAC256_P384_ML-KEM-1024,

KMAC256_ X448_ML-KEM-1024,

KMAC256_ BP-P384_ML-KEM-1024.

8 Hybrid key establishment schemes

8.1 General

8.1.1 Key establishment abstraction

The present clause specifies how to combine two or more key establishment schemes into a hybrid key establishment scheme, under the assumption that all KEM schemes provide at least OW-CPA security.

NOTE: A KEM that provides IND-CPA or IND-CCA also provides OW-CPA. (See annex B for additional details.)

A key establishment mechanism consists of three functions:

- *KeyGen( )*, a key generation function that produces a private key *sk* and public key *P*;
- *ResponseFunc(P)*, a responder function which produces a shared secret *k* and response value *R*, or ⊥ an error indicator;
- *ReceiveFunc(sk, R)*, a receiving function on the private key *sk* and the response value *R* to compute the shared secret *k*, or ⊥ an error indicator.

If *ResponseFunc* returns an error indicator, B shall respond with an error message and terminate the process.

If A receives an error message from B, or if *ReceiveFunc* returns an error indicator, A shall terminate the process.

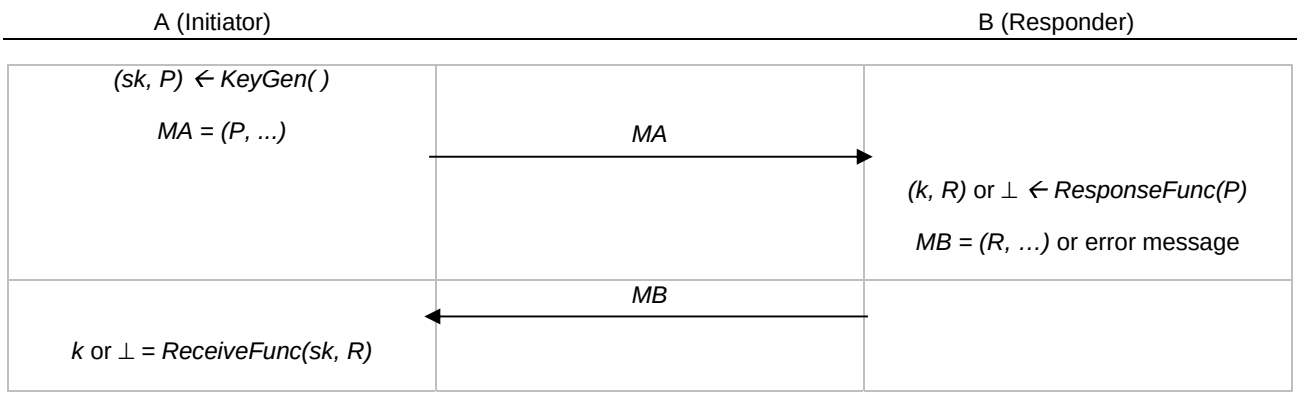

Figure 3: Key establishment abstraction

MA - shall be an octet string containing an encoding of one or more exchanged public keys from Initiator to Responder. *MA* may include session negotiation information.

MB - shall be an octet string containing an encoding of one or more response values. *MB* may include session negotiation information.

Messages shall be protected from unauthorized modification. Messages may be protected using digital signatures from a signing key associated with the sender's identity signed by a trusted third-party certificate authority.

Collectively such an instance of a key establishment is named a *key establishment transaction*.

This is designed to support the major use case of ECDHE with a quantum-safe KEM. Functional mappings between the naming convention and the calling abstractions are provided in clauses 8.1.2 and 8.1.3.

8.1.2 Key establishment abstraction to ECDHE

This clause specifies the mapping of the key establishment abstraction interface to ECDHE:

$$(sk, P) = KeyGen( ) = ECKeyGen( ).$$

The *ResponseFunc* shall be a combination of the *ECKeyGen( )* function and the *ECDH( )* function as follows.

ResponseFunc(P):

1) $(sk', R) = ECKeyGen( )$.
2) k or $\perp = ECDH(sk', P)$.
3) Return (k, R) or $\perp$.

The final *ReceiveFunc* shall be the *ECDH( )* function, k or $\perp$ = *ReceiveFunc(sk, R) = ECDH(sk, R)*, where *R* is the public key (point) of the Responder sent in the message *MB*.

8.1.3 Key establishment abstraction to KEM

This clause specifies the mapping of the key establishment abstraction to a KEM:

$$(sk, P) = KeyGen( ) = KEMKeyGen( ).$$

The *ResponseFunc* shall be the *Encaps( )* function, (k, C) or $\perp$ = *ResponseFunc(P) = Encaps(P)*.

The *ReceiveFunc* shall be the *Decaps( )* function, k or $\perp$ = *ReceiveFunc(sk, C) = Decaps(sk, C)*, where *C* is the ciphertext value, generated by the Responder and sent in the message *MB*.

8.2 Concatenate hybrid key establishment scheme

8.2.1 Concatenate hybrid key establishment scheme - ephemeral

This clause specifies the concatenate hybrid key establishment scheme using ephemeral keys. The key establishment description of Figure 3 is extended to exchange a pair of public keys in a single message and multiple response values in a single message. The concatenate hybrid key establishment scheme is constructed as depicted in Figure 4.

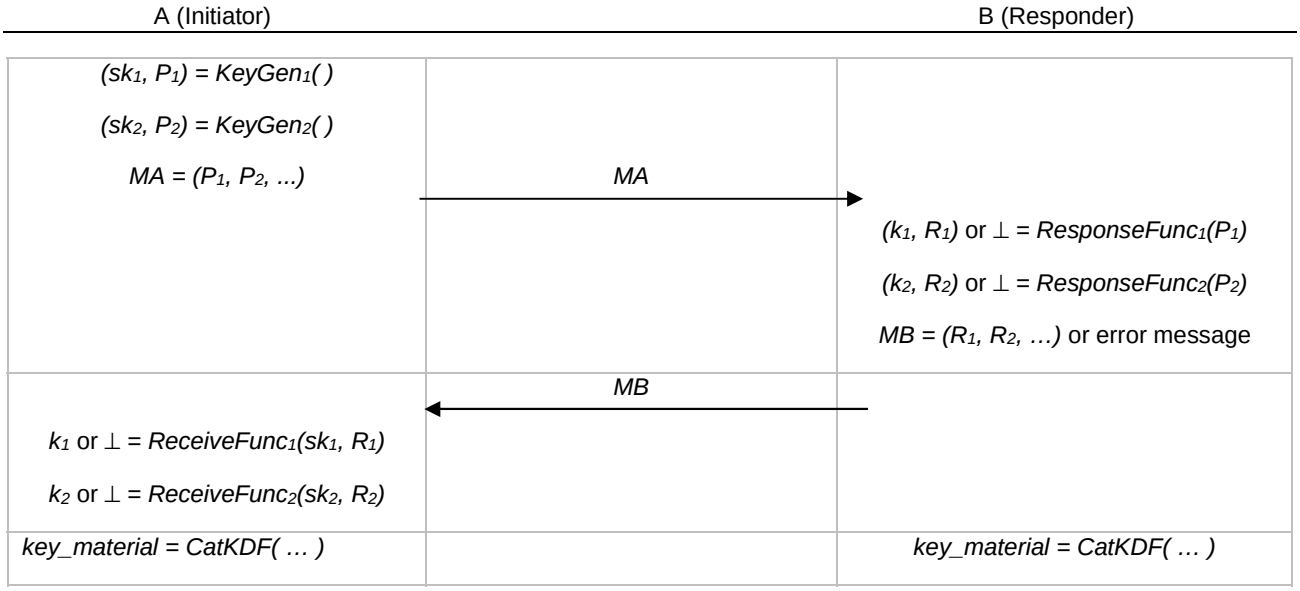

Figure 4: Concatenate hybrid key establishment - ephemeral

If any *ResponseFunc*$_i$ returns an error indicator, B shall respond with an error message and terminate the process.

If *A* receives an error message, *A* shall terminate the process. If any *ReceiveFunc*$_i$ returns an error indicator, *A* shall terminate the process.

MA - shall be an octet string containing an encoding of exchanged public keys P_i from Initiator to Responder. *MA* may include session negotiation information, such as label contribution values.

MB - if *MB* is not an error message, it shall be an octet string containing an encoding of the response values R_i. *MB* may include session negotiation information, such as label contribution values.

Messages shall be protected from unauthorized modification. Messages may be protected using digital signatures from a signing key associated with the sender's identity signed by a trusted third-party certificate authority.

8.2.2 Concatenate hybrid key establishment scheme - static

This clause specifies the concatenate hybrid key establishment scheme using static keys. The key establishment description of Figure 3 is modified to obtain a pair of static public keys in a trusted manner in a single message and a pair of response values in a single message. The roles of Initiator and Responder are replaced with Recipient and Sender. A hybrid key establishment scheme using static keys is constructed as depicted in Figure 5.

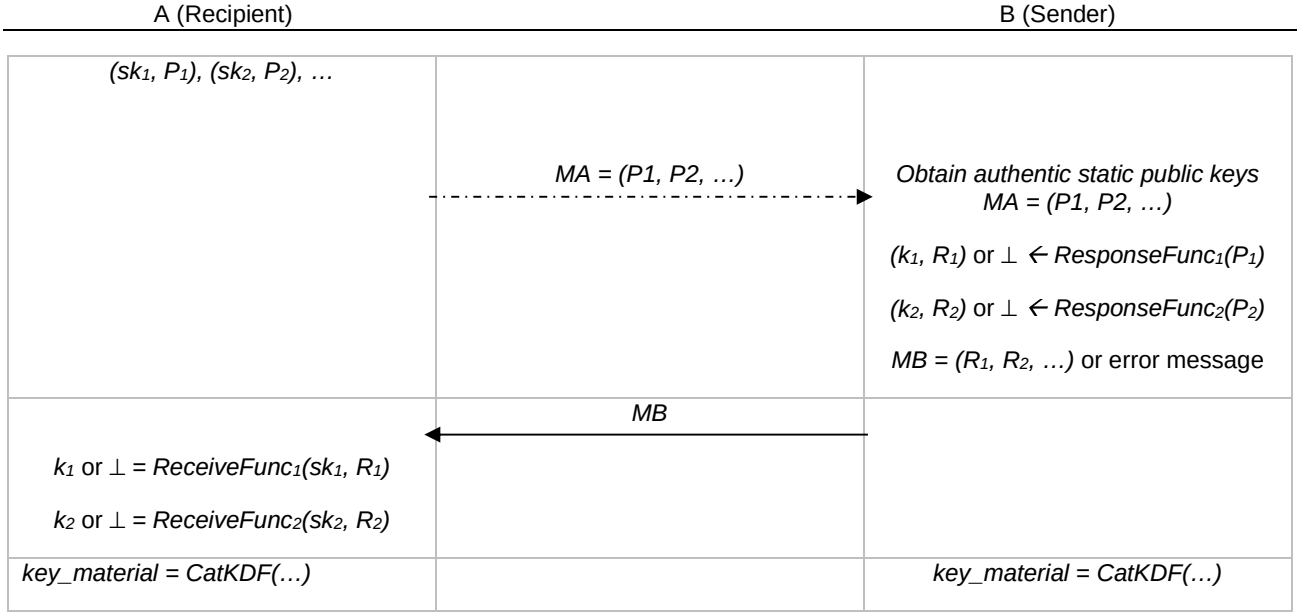

Figure 5: Concatenate hybrid key establishment - static

If any $ResponseFunc_i$ returns an error indicator, B shall terminate the process.

If any $ReceiveFunc_i$ returns an error indicator, *A* shall terminate the process.

MA - Prior to or during the key-establishment, the Sender (B) receives Recipient's (A's) static public keys and additional values such as labels in a trusted manner.

MB - if *MB* is not an error message, it shall be an octet string containing an encoding of the response values R_i. *MB* may include session negotiation information. In the case where more than two key establishment schemes are being used, *MB* shall contain all of the corresponding public keys and ciphertexts.

Messages shall be protected from unauthorized modification. Messages may be protected using digital signatures from a signing key associated with the sender's identity signed by a trusted third-party certificate authority.

8.2.3 Concatenate hybrid key combiner - CatKDF

This clause specifies the concatenate hybrid key combiner using the concatenate KDF CatKDF, as denoted in Figure 4 and Figure 5.

The CatKDF( ) mode shall be defined as follows:

Fixed values:

KDF - The key derivation function being used from clause 7.4.

f - A context formatting function being used from clause 7.2. When KDF is instantiated as HKDF with hash, the underlying hash function is applied to the concatenate-and-hash context formatting function.

k_len - an integer value denoting the length of keys such as *psk*; *k_len* is *digest_len* when CatKDF is instantiated with HKDF or HMAC and *kmac_outlen* when CatKDF is instantiated as KMAC. *kmac_outlen* is set to 32 or 48 octets for KMAC128 and KMAC256, respectively.

Input:

psk - a pre-shared secret key of length *k_len.* It may be present. If not present, this value shall be the empty octet string, ∅.

(k_1, k_2) - *a pair* of octet strings containing shared secrets k_i, derived through a hybrid key establishment, see Figure 4 and Figure 5.

MA, *MB* - octet string of a pair of exchanged messages in establishment of the shared secrets k_i.

info - an optional octet string set by the application as additional information.

label - an optional octet string that specifies a separation of use for the instance of the key establishment. The label values should be constructed from exchanged label contribution values in messages *MA* and *MB*. If a label value is used it shall be of a fixed agreed length. Any labels used in the key derivation function should not be provided as an argument to the same hash function for another purpose in the application.

length - the length in octets of the derived key material *key_material*.

Process:

1) Form $secret = psk \,||\, k_1 \,||\, k_2$.
2) Set $context = f(info, MA, MB)$, where f is a context formatting function.
3) $key\_material = KDF(secret, label, context, length)$.
4) Return *key_material*.

Output:

key_material - derived key material.

An implementation shall fix the order of the shared secrets k_i according to the order of the key establishment schemes defined in clause 7.7.2.

A pre-shared key, *psk*, for this method may be established using key material from a previous session or an alternative key-establishment method like QKD.

8.3 Cascade hybrid key establishment scheme

8.3.1 Cascade hybrid key establishment scheme - ephemeral

This clause specifies the cascade hybrid key establishment scheme using ephemeral keys. The key establishment description of Figure 3 is extended to exchange multiple public keys in different messages and multiple response values in different messages. The cascade hybrid key establishment scheme is constructed as depicted in Figure 6.

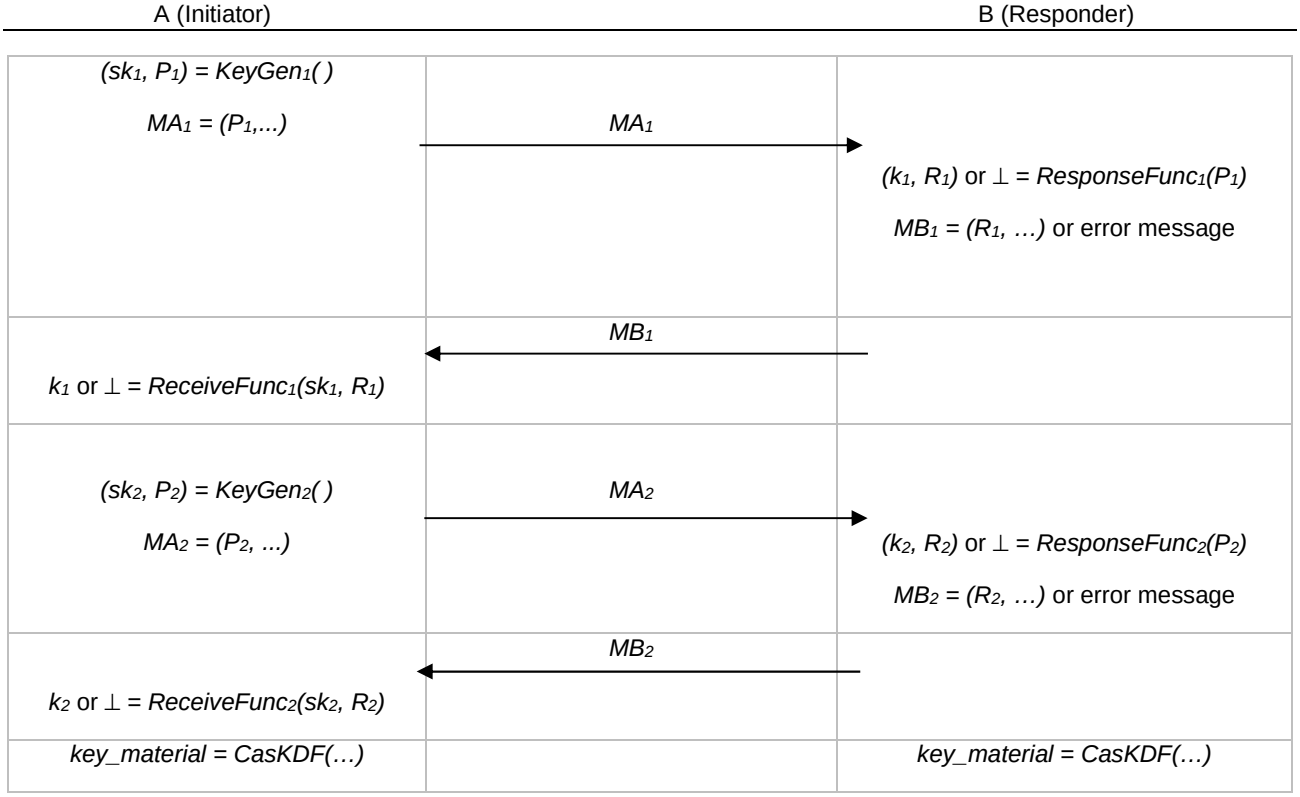

Figure 6: Cascade hybrid key establishment - ephemeral

If any *ResponseFunc*$_i$ returns an error indicator, *B* shall return an error message and terminate the process.

If *A* receives an error message from *B*, *A* shall terminate the process. If any *ReceiveFunc*$_i$ returns an error indicator, *A* shall terminate the process.

MA$_i$ - shall be an octet string containing an encoding of exchanged public key P_i from Initiator to Responder. *MA*$_i$ may include session negotiation information, such as label contribution values.

MB$_i$ - shall be an octet string containing and encoding of the response value R_i. *MB*$_i$ may include session negotiation information, such as label contribution values.

Messages shall be protected from unauthorized modification. Messages may be protected using digital signatures from a signing key associated with the sender's identity signed by a trusted third-party certificate authority.

8.3.2 Cascade hybrid key establishment scheme - static

This clause specifies the cascade hybrid key establishment scheme using static keys. A hybrid key establishment scheme using static keys is constructed as depicted in Figure 7.

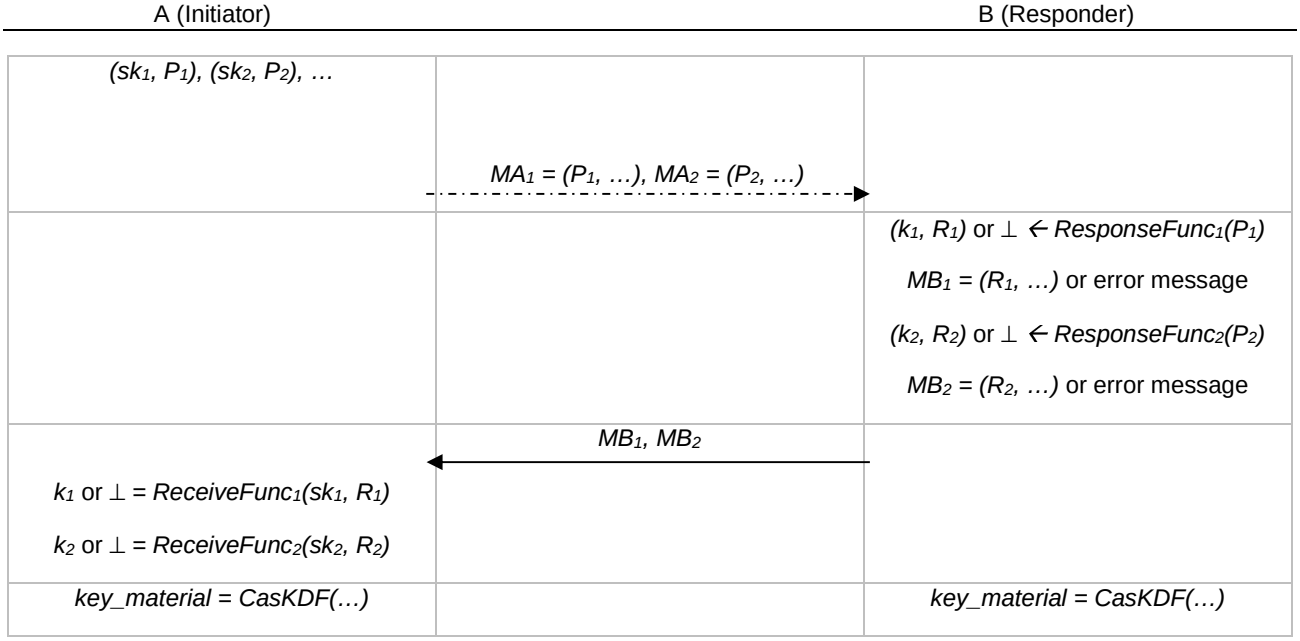

Figure 7: Cascade hybrid key establishment - static

If any $ResponseFunc_i$ returns an error indicator, B shall terminate the process.

If any $ReceiveFunc_i$ returns an error indicator, *A* shall terminate the process.

MA_i - Prior to or during the key-establishment, the Sender (B) receives Recipient's (A's) static public keys and additional values such as labels in a trusted manner.

MB_i - if MB_i is not an error message, it shall be an octet string containing an encoding of the response values R_i. MB_i may include session negotiation information such as label contribution values.

Messages shall be protected from unauthorized modification. Messages may be protected using digital signatures from a signing key associated with the sender's identity signed by a trusted third-party certificate authority.

8.3.3 Cascade hybrid key combiner - CasKDF

This clause specifies the cascade hybrid key combiner using the cascade KDF CasKDF(), as denoted in Figure 6 and Figure 7.

The CasKDF() mode shall be defined as follows:

Fixed values:

KDF - the key derivation function being used from clause 7.4.

PRF - the PRF function being used from clause 7.3. When KDF is instantiated as HKDF or HMAC with hash, the underlying hash function is applied to the concatenate-and-hash context formatting function.

k_len - an integer value denoting the length of intermediate keys such as *psk*, $chain\_secret_i$, and $round\_secret_i$; *k_len* is *digest_len* when CasKDF is instantiated with HKDF or HMAC and *kmac_outlen* when CasKDF is instantiated as KMAC. *kmac_outlen* is set to 32 or 48 octets for KMAC128 and KMAC256, respectively.

Input:

psk - a pre-shared secret key of length *k_len*. It may be present. If not present this value shall be the empty octet string, ∅.

(k_1, k_2) - a pair of octet strings containing shared secrets k_i, derived through key establishment mechanisms, see Figure 6 and Figure 7.

MA_i, MB_i - octet strings of exchanged messages in establishment of the shared secret k_i, for each $i = 1,2$.

$info_i$ - an optional octet string set by the application as additional information to establish secret *i*.

$label_i$ - an optional octet string that specifies a separation of the instance of the key establishment. The label value should be constructed from exchanged label contribution values in messages MA_i and MB_i. If a label value is used it shall be of a fixed agreed length. Any labels used in the key derivation function should not be provided as an argument to the same hash function for another purpose in the application.

$length_i$ - a length value for intermediate and final key material for $i = 1, 2$.

Process:

1) $chain\_secret_0 = psk$, a *k_len* value or the empty string, ∅.
2) For $i = 1, 2$.
   a) $round\_secret_i = PRF(chain\_secret_{i-1}, k_i, MA_i, MB_i)$.
   b) $chain\_secret_i \,||\, key\_material_i = KDF(round\_secret_i, label_i, info_i, k\_len + length_i)$.
3) Return $chain\_secret_1$, $chain\_secret_2$, and $key\_material_1$, $key\_material_2$.

Output:

$key\_material_1$ and $key\_material_2$ - a pair of intermediate and final keys of length $length_1$ and $length_2$.

$chain\_secret_1$ and $chain\_secret_2$ - a pair of chain secrets of length *k_len*.

The KDF is run twice, with each time injecting the shared secret from the next key establishment.

The intermediate key material, $key\_material_1$ may be used in the exchange of messages MA_2 *and* MB_2.

Implementations shall fix the order of key establishment schemes in the exchanged messages in MA_i and MB_i according to the order of the key establishment schemes defined in clause 7.7.2.

A pre-shared key, *psk*, for this method may be established using a key material from a previous session or an alternative key-establishment method like QKD.

Annex A (informative): Background

A.1 Quantum computing threats to traditional key exchange protocols

Quantum computing, first conjectured by Richard Feynman in 1982, has progressed to small-scale limited quantum computing. In contrast to the classical computing paradigm, where the basic computational unit is the "bit" - a two-level system which can hold either a value of '0' or '1', quantum computing represents a new paradigm of computation which harnesses the fundamental laws of quantum mechanics to perform computations on basic units called "qubits" - two-level *quantum mechanical* systems.

The laws of quantum mechanics are strikingly different from the familiar laws of Newtonian mechanics. For example, qubits are allowed to be in linear "superpositions" of the 0 and 1 states, in which the qubit can be seen as "holding both the value at '0' and '1' at the same time". Such an analogy is not precisely correct, as coherent superpositions cannot be really interpreted as a quantum system having two states at once; quantum superpositions are an intrinsic feature of quantum mechanics with no direct analog in classical physics. Note also that quantum superpositions are fundamentally different from classical fuzzy states in which a bit can be in the state '0' with probability p and state '1' with probability 1-p. Qubits can also "interfere" with each other, in a way that is similar to how waves interfere. Moreover, two or more qubits can be "entangled" with each other, i.e. correlated stronger than any classical systems can ever be. The quantum state of n qubits is described by 2^n complex numbers (amplitudes). Therefore, classically simulating n qubits requires in general an exponential amount of storage (and time) and is not feasible for large n. Quantum computers take advantage of quantum mechanical laws and features of superposition, interference and entanglement to manipulate the state of n qubits "at once", allowing for novel *quantum algorithms* that can, in some cases, provide significant speedups when compared to their classical counterparts.

In 1994, Peter Shor showed how to utilize Simon's Algorithm [i.5] for the hidden subgroup problem, to factor large semiprimes and solve the discrete log problem [i.6]. Shor's algorithm effectively breaks finite field and elliptic curve discrete-log-based, and integer-factorization-based cryptosystems, like ECC, and RSA. However, a large-scale, fault-tolerant quantum computer would be required to use Shor's algorithm to break cryptographically relevant instances of ECC and RSA. There are a number of challenges in building one. While progress is routinely made on these challenges, it is far from certain that a large-scale fault-tolerant quantum computer will be constructed.

Cryptographic engineering is fundamentally based on understanding the probability of a break in a cryptographic primitive and weighing that against the value of the information that primitive is used to protect. For instance, the probability that an adversary can guess a 128-bit key is $1/2^{128}$, and that information might protect the confidentiality of an individual's credit card information during a financial transaction.

Many encrypted communications are negotiated using asymmetric key establishment agreement schemes like ECC and RSA, e.g. TLS, IPSec, SSH. When viewed through the lens of traditional cryptographic engineering the probability of a large-scale quantum computer being available during the confidentiality lifespan requirement of these sessions exceed the probability that many organizations have historically been willing to tolerate.

Similar to the transition from RSA to ECDHE that provided perfect forward secrecy, quantum-safe KEMs promise to deliver *quantum forward secrecy* - the property of a key establishment protocol that gives assurances the session keys will not be compromised even if the adversary has a large-scale fault tolerant quantum computer.

A.2 Rationale for quantum-safe hybrid key establishment

Today, the existing key exchanges are at risk from a future adversary with a quantum computer. Assurances that the best-known attacks on the proposed quantum-safe KEMs are not sufficient to warrant sole reliance on for confidential communication channels. A critical factor in evaluating the security of a cryptographic primitive is not solely the complexity of the best-known attack and the probability of its success, it is also the assurance that no other attack will be discovered. The cycle of analysis, publish, disseminate required to build assurance for new cryptographic schemes cannot be done in parallel and subsequently takes time for this cycle to repeat.

The proposed post-quantum KEMs have not reached a level of assurance or community agreement that warrants a final selection for standardization and reliance as a single technique to provide confidentiality. Quantum-safe hybrid key establishment schemes pair a high-assurance but quantum-vulnerable key exchange with a potentially quantum-safe key encapsulation mechanism to obtain the best possible key exchange. In addition to providing potential quantum-safety this approach allows for the design of new cryptographic protocols and applications that are tuned to the new bandwidth and computational requirements of quantum-safe key establishment.

Annex B (informative): Security consideration

B.1 Security definitions

A key exchange scheme is secure according to Indistinguishability under Chosen-Plaintext Attack (IND-CPA) if an adversary that is given all of the public information from an exchange (e.g. MA, MB) is unable to distinguish the resulting key from a key selected independently at random. A key exchange scheme is One Way under Chosen-Plaintext Attack (OW-CPA) if an adversary that is given all of the public information from an exchange (e.g. MA, MB) is unable to produce the resulting key. These definitions apply to computationally bounded adversaries, and the adversary is allowed to succeed with some small probability.

The CatKDF and CasKDF schemes are IND-CPA secure in the random oracle model as long as at least one underlying scheme is OW-CPA. However, for use cases that require a static key a stronger security property, Indistinguishability under Chose-Cipher Attacks (IND-CCA), is required.

A key exchange scheme is IND-CCA if the scheme is IND-CPA and that the adversary is given access to a decryption oracle on any ciphertexts except one associated with the challenge ciphertext and is unable to distinguish the resulting key from a key selected independently at random. Similarly, a scheme is One Way under Chosen Ciphertext Attack (OW-CCA) if the adversary has access to the decryption oracle on any ciphertext except one associated with the challenge ciphertext and is unable to produce the resulting key.

The CatKDF schemes are IND-CCA secure in the random oracle model as long as at least one of the underlying schemes is One Way under Chosen-Ciphertext Attacks (OW-CCA).

There are additional security properties that may be desirable for a key exchange scheme, for instance, key compromise impersonation security or unknown key share security. The present document makes no security assertions beyond the IND-CPA and IND-CCA security in the random oracle model. In this regards, a prudent interpretation of the hybrid key exchanges defined in the present document is to consider them as hybrid KEMs.

For detailed security definitions, and proofs, see [i.11] and [i.12]. Alternate constructions suitable for TLS with security proofs can be found in [i.3].

Annex C (informative): Message Encoding for Test Vector Generation

C.1 Message Formatting Function for Test Vector Generation

This clause defines a message formatting function. It is the formatting used to generate the test vectors in Annex D. The message formatting function takes a ciphersuite identifier (cid), and an ordered sequence of octet strings, and converts them into a delimited octet string. The message construction function has the following calling interface:

$$message = m\_format(cid, val_1, val_2, \ldots)$$

where the parameters and output are defined as follows.

Input:

cid - ciphersuite identifier of length 16 bits consisting of 4-bit substrings as specified in clause 6.2.3; the most significant 4 bits $cid_{0, .., 3}$ denote the underlying KDF from clause 7.4, $cid_{4, .., 7}$ denote the underlying classical ECDH key establishment instance from clause 7.5.2 , and $cid_{8, .., 11}$ denote the underlying post-quantum ML-KEM instance from clause 7.6.2. $cid_{12, .., 15}$ denotes concatenate CatKDF from clause 8.2 or cascade CasKDF from clause 8.3 operation mode.

A 16-bit *cid* is subdivided into 4-bit substrings CID_0, CID_1, CID_2, CID_3 so if the value is:

$$CID = 0x0123 = (0000000100100011)_{bin}$$

then:

$CID_0 = (0000)_{bin}$, denotes the KDF instance as from clause 7.4
$CID_1 = (0001)_{bin}$, denotes elliptic curve ECDH from clause 7.5.2
$CID_2 = (0010)_{bin}$, denotes post quantum KEM from clause 7.6.2
$CID_3 = (0011)_{bin}$, denotes concatenate CatKDF from clause 8.2 or cascade CasKDF from clause 8.3

Our test vectors use the following CID values:

enum CID_0 = {HKDFwSHA256 = 1, HKDFwSHA384 = 2, HMACwSHA256 = 4, HMACwSHA384 = 5, KMAC128 = 7, KMAC256 = 8};

enum CID_1 = {P256 = 1, P384 = 2, BP-P256 = 4, BP-P384 = 5, X25519 = 7, X448 = 8};

enum CID_2 = {ML-KEM-512 = 1, ML-KEM-768 = 2, ML-KEM-1024 = 3};

enum CID_3 = {CatKDF = 1, CasKDF = 2};

$val_1, val_2, \ldots$ - an ordered sequence of octet strings each of length less than 2^{32} octets, where $n > 0$.

Process:

1) Set *buffer = cid*.
2) For $i = 1, \ldots, n$.
   a) Set $len_i = len(val_i)$, returns the length of val_i in octets.
      i) If $len_i > 2^{32} - 1$, return error.
      ii) $L_i = [len_i]_{32}$ - a 32-bit integer value expressed as 4 octets.
   b) Set $buffer = buffer \,||\, L_i \,||\, val_i$.
3) Set *message = buffer*.
4) Return *message*.

Output:

message - an octet string representing the message value or an error.

The message formatting function is designed to be used in conjunction with concatenate and cascade hybrid key establishment scheme combiners.

In CatKDF, the values *labelA, P_1, P_2* are mapped to *val_1*, *val_2*, and *val_3* in a single *m_foramt* invocation as follows:

$$MA = m\_format(cid, labelA, P_1, P_2).$$

Similarly, *MB* is formed as:

$$MB = m\_format(cid, labelB, R_1, R_2).$$

In CasKDF, the values $labelA_i$ and P_i are mapped to val_1 and val_2 in separate *m_foramt* invocations, for each $i = 1, 2$, as follows:

$$MA_i = m\_format(cid, labelA_i, P_i).$$

Similarly, MB_i is formed as:

$$MB_i = m\_format(cid, labelB_i, R_i).$$

NOTE 1: In the static hybrid key establishment scheme, prior to or during the key-establishment, the Sender (B) receives Recipient's (A's) static public keys in a trusted manner.

NOTE 2: Any application of this formatting function should use a fixed number and fixed order of arguments to avoid potential parsing issues.

NOTE 3: The test vector application uses the label contribution values $label_A$ and $label_B$ as *k_len* random nonces, and constructs a label for the key derivation function by bytewise XORing these values together $label = label_A \wedge label_B$.

Annex D (informative): Test Vectors

D.1 Test Vectors Repository

Test vectors are generated by the C reference implementation for Quantum-safe Hybrid Key Exchanges that can be found at: https://forge.etsi.org/rep/cyber/103744_QHKEX/-/tree/kats_v1.2.x.

This code is provided as an informative implementation of the Quantum-safe Hybrid Key Exchanges for the Concatenate KDF (CatKDF) and Cascade KDF (CasKDF).

The code is not intended for production use. It is intended to be a reference implementation for test vectors for the present document.

Annex E (informative): Bibliography

- NIST CAVP SP 800-56A: "ECC CDH Primitive Test Vectors".
- IETF RFC 8734 (2020): "Elliptic Curve Cryptography (ECC) Brainpool Curves for Transport Layer Security (TLS) Version 1.3".
- Kwiatkowski K. (2024): "Post Quantum Cryptography KATs".

Annex F (informative): Change history

Date	Version	Information about changes
January 2022	V1.1.5	Updated to include static-ephemeral use case, and an abstraction of the context formatting function.
March 2022	V1.1.6	Minor edits based on working group feedback.
September 2022	V1.1.7	Aligned draft to latest NIST selection, removed reference to schemes no longer being considered for NIST standardization.
May 2023	V1.1.8	Removed a context formatting function type because of incompatibility with current implementations. Updated references.
September 2023	V1.1.9	Added latest references for FIPS 203 (draft) and Hybrid KEM proof paper. Updated references through out the document. Security consideration is augmented to include a paragraph about IND-CCA for CatKDF.
February 2024	V1.1.10	Added description of KMAC as a KDF function.
September 2024	V1.1.11	Added in fixed parameter sets for curves and KEMs. Updated specification to restrict KEMs to FIPS 203 ML-KEM standard. Removed NIST P521 and brainpool curves. Restored the previous concatenate-based formatting function, and restored the PRF to KMAC mapping.
December 2024	V1.1.12	Refined the fixed parameter sets. Re-ordered some of the sections to have a single description for CasKDF and CatKDF. Added an Annex C on message formatting used for test vectors. Small modification for CasKDF to accommodate KMAC implementation detail.
January 2025	V1.1.13	Added HMAC KDF, added error conditions to formatting functions, fixed label lengths when used, strengthened the language around authenticating messages, and fixed the number of key establishment schemes and order for the shared secrets.
February 2025	V1.1.14	Addressed comments received from on V.1.1.13 (removing "shall" statements from notes, adding motivation for KDFs, and adding missing HMAC reference as a KDF).
August 2025	V1.2.2	Update test vectors repository to point to forge project.

History

Version	Date	Status
V1.1.1	December 2020	Publication
V1.2.1	March 2025	Publication
V1.2.2	February 2026	Publication